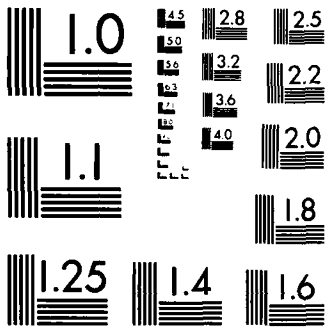
AD-A164 180 IMAGE SEGMENTATION USING THE MILITARY SPECIFICATION 1750A MICROPROCESSOR(U) NAVAL POSTGRADUATE SCHOOL MONTEREY CA P D CODY DEC 85 1/1

UNCLASSIFIED F/G 9/2 NL


1.0
4.5
5.0
5.6
6.3
7.1
8.0
2.8
3.2
3.6
4.0
2.5
2.2
2.0
1.1
1.8
1.25
1.4
1.6

2

NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

IMAGE SEGMENTATION USING THE MILITARY SPECIFICATION 1750A MICROPROCESSOR

by

Percy Dean Cody III

December 1985

Thesis Advisor: C. H. Lee

Approved for public release; distribution is unlimited.

SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED	1b. RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution is unlimited.
2b. DECLASSIFICATION / DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S)	5. MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION Naval Postgraduate School; 6b. OFFICE SYMBOL (If applicable) 62	7a. NAME OF MONITORING ORGANIZATION Naval Postgraduate School
6c. ADDRESS (City, State, and ZIP Code) Monterey, California 93943-5100	7b. ADDRESS (City, State, and ZIP Code) Monterey, California 93943-5100
8a. NAME OF FUNDING/SPONSORING ORGANIZATION; 8b. OFFICE SYMBOL (If applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
8c. ADDRESS (City, State, and ZIP Code)	10. SOURCE OF FUNDING NUMBERS: PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO. / WORK UNIT ACCESSION NO.

11 TITLE (Include Security Classification)
IMAGE SEGMENTATION USING THE MILITARY SPECIFICATION 1750A MICROPROCESSOR

12 PERSONAL AUTHOR(S)
Cody, Percy Dean III

13a TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
Master's Thesis	FROM ________ TO ________	1985 December	81

16 SUPPLEMENTARY NOTATION

17 COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	1750A/F9450 Microprocessor, Image Segmentation

19 ABSTRACT (Continue on reverse if necessary and identify by block number)

The use of digital computers to process various types of sensor date is becoming increasingly common, in both civilian and military applications. One example of this use is the enhancement of photographs to increase their clarity, or emphasize a particular detail.

Previously, the computers used to perform this processing was done in specialized circuits, mainframe or minicomputers. More recently, extremely powerful microprocessors have become available that show potential to be applied in this area.

This thesis explores a particular class of image processing, known as Image Segmentation, implemented on a particular microprocessor. The microprocessor is the Fairchild F9450, the first civilian version of the 1750A military specification microprocessor.

20 DISTRIBUTION/AVAILABILITY OF ABSTRACT [✓ UNCLASSIFIED/UNLIMITED] ☐ SAME AS RPT. ☐ DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION UNCLASSIFIED
22a. NAME OF RESPONSIBLE INDIVIDUAL Prof C. H. Lee	22b. TELEPHONE (Include Area Code) 408-646-2190; 22c. OFFICE SYMBOL 62Le

DD FORM 1473, 84 MAR   83 APR edition may be used until exhausted. All other editions are obsolete.   SECURITY CLASSIFICATION OF THIS PAGE

This microprocessor, along with its associated chip set, appears well suited to image processing, having high speed capability, direct floating point arithmetic instructions, multiprocessing capacity, and the ability to address up to sixteen megabytes of memory.

Additionally, a sophisticated software development tool set, known as Microprocessor Pascal, is available to develop and test software for the 1750A/F9450 microprocessor. This tool set allows software to be developed on the VAX-11/780 minicomputer, targeted for final use on the 1750A/F9450.

This work utilized the Microprocessor Pascal tool set to test and compare representative Image Segmentation algorithms. The speeds of execution and code sizes of the programs were determined for the F9450/1750A microprocessor and the VAX-11/780 minicomputer, and were compared to determine the feasibility of using the F9450/1750A microprocessor for image segmentation work.

Several images resulting from the image segmentation processing are included, as well as the Pascal programs used to perform the processing.

Image Segmentation using the Military Specification 1750A Microprocessor

by

Percy D. Cody III
Lieutenant, United Stated Navy
B.S.E.E. University of Kansas, December 1980

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN ELECTRICAL ENGINEERING

from the

NAVAL POSTGRADUATE SCHOOL
December, 1985

Author: Percy D. Cody III
Percy D. Cody III

Approved by: [illegible]
Chin-Hwa Lee, Thesis Advisor

alex Gerba Jr.
Alex Gerba Jr.,Second Reader

[illegible]
Harriett B. Rigas
Chairman, Department of Electrical
and Computer Engineering

JN Dyer
John N. Dyer
Dean of Science and Engineering

ABSTRACT

The use of digital computers to process various types of sensor data is becoming increasingly common, in both civilian and military applications. One example of this use is the enhancement of photographs to increase their clarity, or emphasize a particular detail.

Previously, the computers used to perform this processing was done in specialized circuits, mainframe or minicomputers. More recently, extremely powerful microprocessors have become available that show potential to be applied in this area.

This thesis explores a particular class of image processing, known as Image Segmentation, implemented on a particular microprocessor. The microprocessor is the Fairchild F9450, the first civilian version of the 1750A military specification microprocessor.

This microprocessor, along with its associated chip set, appears well suited to image processing, having high speed capability, direct floating point arithmetic instructions, multiprocessing capacity, and the ability to address up to sixteen megabytes of memory.

Additionally, a sophisticated software development tool set, known as Microprocessor Pascal, is available to develop and test software for the 1750A/F9450 microprocessor. This tool set allows software to be developed on the VAX-11/780 minicomputer, targeted for final use on the 1750A/F9450.

This work utilized the Microprocessor Pascal tool set to test and compare representative Image Segmentation algorithms. The speeds of execution and code sizes of the programs were determined for the F9450/1750A microprocessor and the VAX-11/780 minicomputer, and were compared to determine the feasibility of using the F9450/1750A microprocessor for image segmentation work.

Several images resulting from the image segmentation processing are included, as well as the Pascal programs used to perform the processing.

TABLE OF CONTENTS

LIST OF FIGURES

ACKNOWLEDGEMENTS

I wish to gratefully acknowledge my thesis advisor, Professor Chin-Hwa Lee, who provided invaluable assistance in completion of this thesis.

I would also like to express my gratitude to Professor Alex Gerba Jr. for his assistance.

I. INTRODUCTION

A. GENERAL

The application of image processing is expanding into many new areas including the military. In many cases the need exists to enhance a desired image often in the presence of background clutter, to allow target identification, etc. This is often done by an automated system. One type of such processing is the method of Image Segmentation.

Image Segmentation involves the conversion of an image with multiple levels of gray values (which can represent color, brightness, or infrared radiation as examples) into a "binary" image, which has only two levels. This has the effect of converting a "half tone" image into a "black and white" one. Figure 1, as an example, shows the input and output images from an Image Segmentation system. The input is a ship image composed of pixels which vary over a range from zero to two hundred and fifty five. The output image is the same ship where the image pixels have only two values; zero and one. This process has the additional effect of removing a great deal of the background clutter.

Like most computer graphics applications, image segmentation is a very "CPU intensive" process; requiring a large amount of computation. Performing this type of processing in real time will require very high speed in both hardware and software. For this reason such processing is often done with specialized, custom designed hardware. In this study, the possibility of efficiently performing image segmentation with a standard, general purpose microprocessor is explored.

B. PURPOSE OF WORK

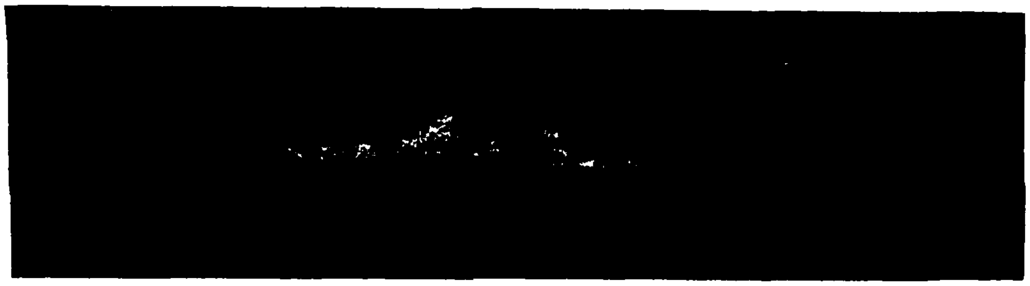
The purpose of this thesis is to determine and compare the speed of image segmentation in two different computer architectures: the 1750A/F9450 microprocessor, and the VAX-11/780 minicomputer using the VMS operating system. Comparisons will be made in terms of actual speeds of execution, sizes of generated code, and overall efficiency. From these factors, it should be possible to determine which method is more appropriate for a given application. While the images used for this work are infrared images of ships, the techniques used are applicable to a wide range of applications and sensor types, including areas such as geological surveys by aerial photography, medical imaging, and so forth.

It should also be noted that, while a particular microprocessor and software development system is used here for this work, it is not a unique selection, and other combinations of tools could be equally applicable.

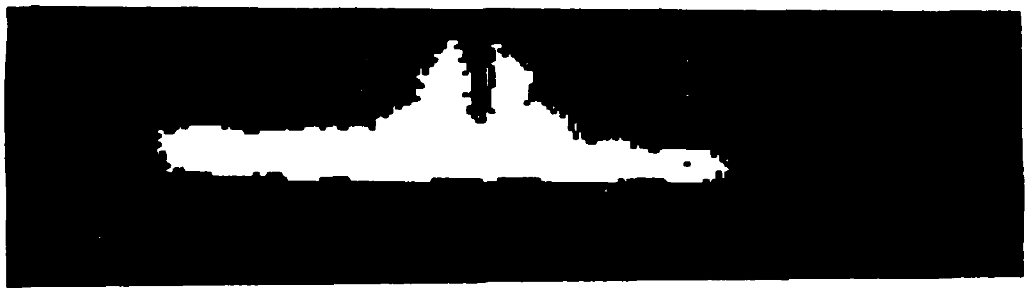
This work will present a description of the hardware and software used, give a brief discussion of two representative image segmentation algorithms, and present the results of the comparison between the two algorithms.

It was necessary to determine the 1750A/F9450 CPU operating speeds indirectly (for reasons to be discussed). The method which was derived for doing this will also be explained and demonstrated.

Finally, the results derived will be analyzed, and a rational to explain them will be discussed relating to the actual merits of the 1750A/F9450 microprocessor, versus the VAX-11/780, for image segmentation processing.



Before Processing:



After Processing:

Figure 1. Ship Images Before and After Segmentation

It is discovered that, in general, the 1750A/F9450 microprocessors are capable of performing image segmentation efficiently, but not normally fast enough for real time operations, unless certain special methods are used.

Possible methods of increasing the speed of operation are presented.

II. 1750A ARCHITECTURE AND SOFTWARE TOOL SET

A. HARDWARE REQUIREMENTS

As stated earlier, virtually any type of image processing makes great demands on the hardware and software used. Image segmentation is no exception.

The first hardware requirement is the need to store and process relatively large amounts of data. This results from the fact that images require at least two dimensional data arrays, and each pixel requires enough bits to represent the desired number of intensity levels in the image. In some cases the large amount of memory required may be reduced somewhat by means of efficient algorithms which require only a small portion of an image to be processed at a time (through such means as "overlap and save" methods of convolution). In general however, the trend is towards "real time" systems with large capacity, such as operator displays in aircraft and medical imaging systems.

The second hardware requirement is for the processor to operate at sufficiently high speeds to meet the design needs. If the processor is to analyze only off-line data, the speed requirement is not as great. Many military and industrial systems however, often require the processing work to be done in real time. This creates the need for a microprocessor to operate at higher speeds than those previously available.

B. 1750A/F9450 MICROPROCESSOR

The microprocessor and software development tool set selected for this work are capable of supporting the memory and speed requirements just stated.

The microprocessor selected for use is the Fairchild F9450 16-bit microprocessor, which is a civilian version of the military 1750A microprocessor. The programmer's register diagram of the 1750A/F9450 is shown in Figure 2 [Ref. 1]. The block diagram of the actual chip architecture is illustrated in Figure 3 [Ref. 2].

As illustrated in Figure 3, the 1750A microprocessor architecture has five sections: data processor, microprogrammed control, address processor, interrupt and fault processor, and timing unit. The data processor allows use of a variety of data types and direct floating point operations instructions. The address processor uses an independent incrementer for the Instruction Counter, and also allows a wide range of addressing modes for the microprocessor. The interrupt processor and timing units are especially useful for multiprocessor operations, as will be discussed later.

The architecture is similar in overall conception to the VAX-11/780, but lacks some features. One example is the lack of a separate numeric coprocessor, similar to the VAX's Floating Point Accelerator option. Additionally, the 1750A/F9450 lacks any built-in facilities for direct implementation of "virtual memory."

The 1750A/F9450 instruction set has a number of instructions to make use of its powerful architecture. Among these are instructions to control the two on- chip timers, and a Built -In Function to allow the direct use of user defined instructions.

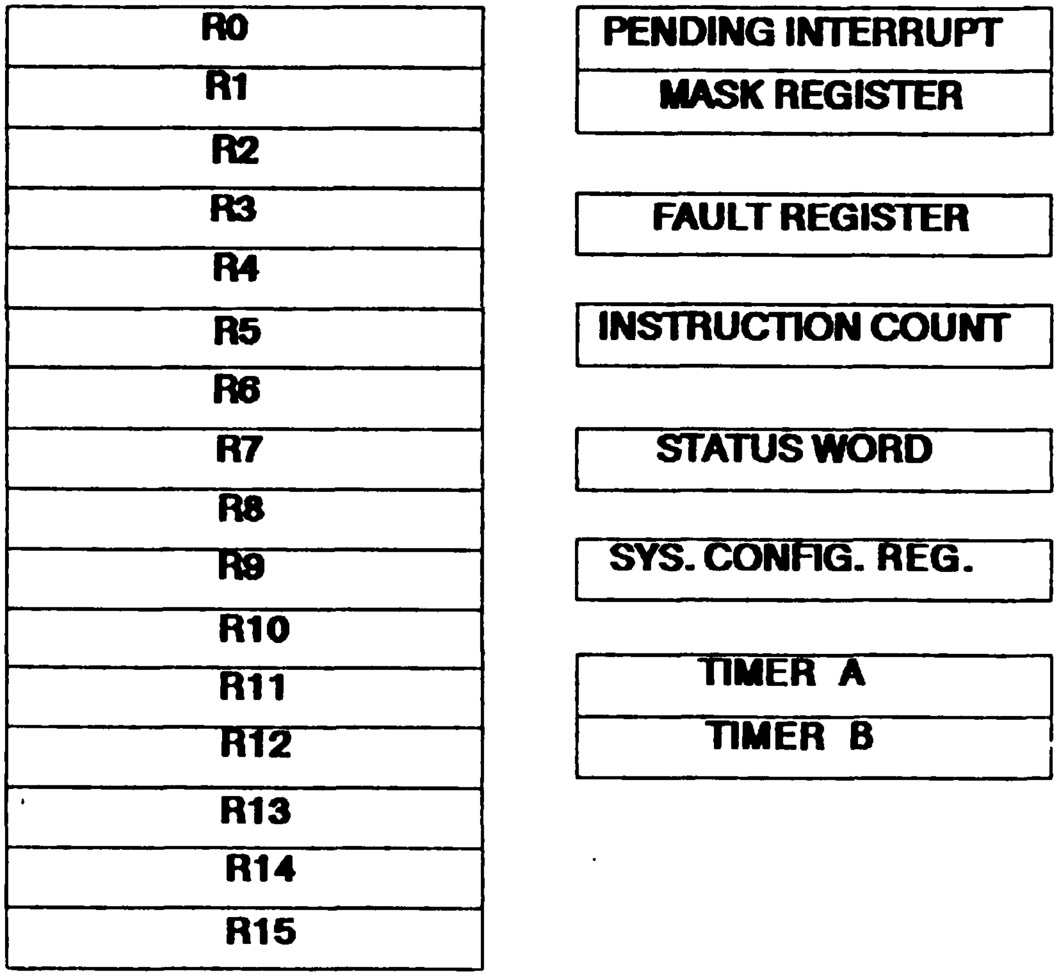


Figure 2. Programmer's Register Model of 1750A/F9450

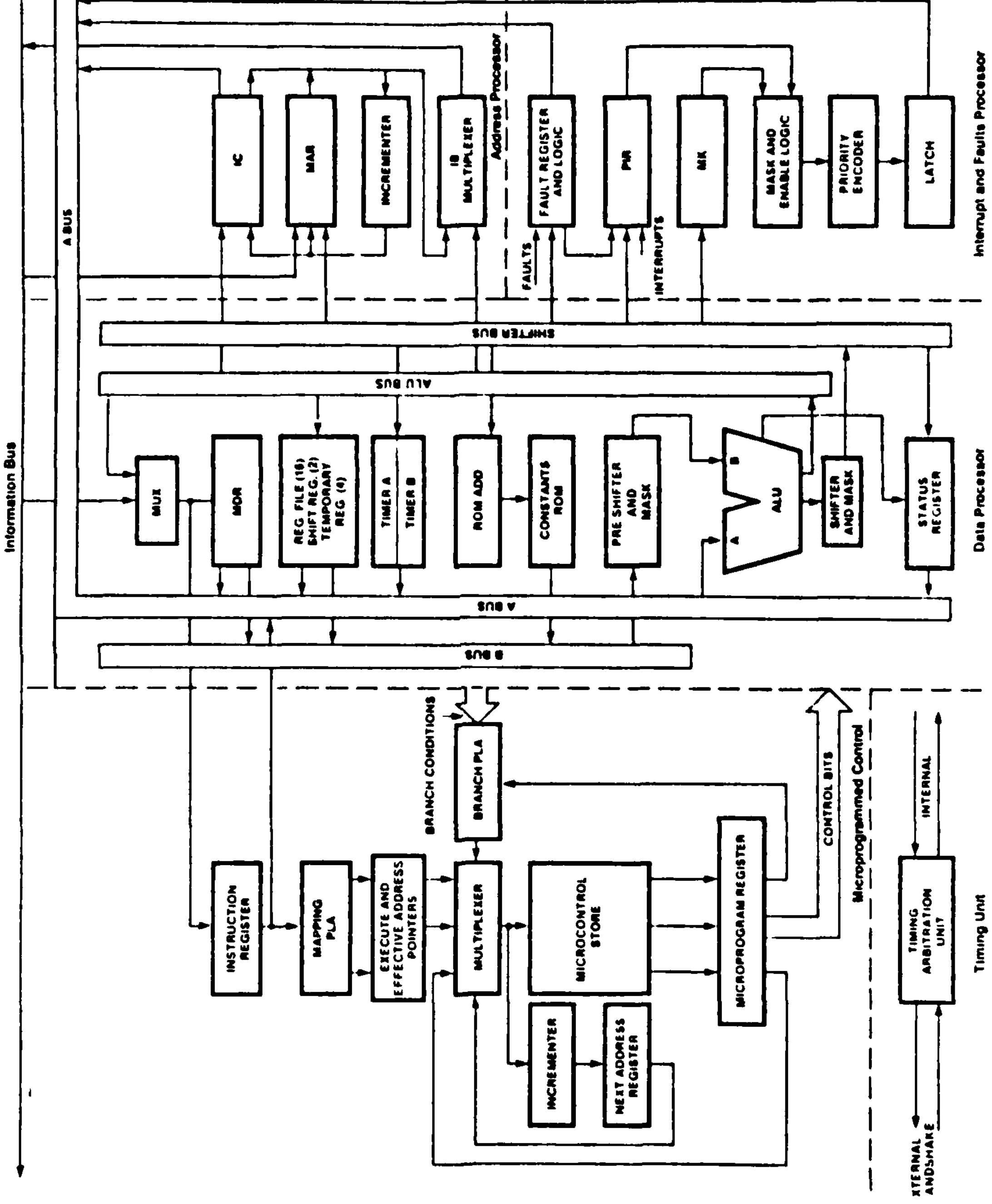


Figure 3. F9450/1750A Microprocessor Architecture

The 1750A/F9450 CPU is a highly sophisticated microprocessor, as can be seen in Figures 2 and 3. It includes sixteen 16 bit general purpose registers, a 16 bit Status Word register and a System Configuration Register in its internal architecture. The general purpose and Status Word registers are very similar in concept to the VAX-11/780 architecture, which uses sixteen 32 bit general purpose registers, and a 32 bit Processor Status Longword register. (Of course, 32 bits allow a greater range of instructions, and greater data accuracy. The architecture itself however, is quite similar.) The 1750A/F9450 System Configuration Register contains information relating to the chip's external environment, such as the presence or absence of an additional microprocessor, memory protection unit, or block protection unit, and the interrupt mode in use. The VAX system doesn't use a configuration register, and it is normally installed in a more standardized configuration.

The 1750A/F9450 CPU is capable of operating at clock speeds of up to twenty megahertz. This microprocessor is one component of a chip set which also includes a Memory Management Unit (the F9451) and a Block Protect Unit (the F9452). Alone, the microprocessor is capable of addressing up to two million 16 bit words of random access memory, and up to twenty million words with the Memory Management Unit. The 1750A/F9450 is highly optimized for real time operation. The features to achieve this capability include a sophisticated 16 vector interrupt handling system, built-in multiprocessor capabilities, and two programmable timers on the chip. This microprocessor also features 32 and 48 bit floating point arithmetic, built in self-test upon power up or reset, and fault handling capabilities. This architecture is highly advanced for a microprocessor, but it is not comparable in overall capability to a powerful minicomputer system such as the VAX system, which is the architecture for comparison, as the VAX system is designed for multi user/timesharing systems.

One of the significant differences between the two systems is the size of the assembly language instruction sets. The 1750A/F9450 has 141 instructions in its set, while the VAX has over 240. This greater flexibility should enable a VAX compiler to convert a high level language statement into a lesser number of assembly language statements than the 1750A/F9450 compiler would require. Another advantage of the VAX system, is a richer range of addressing modes. This will be discussed later in this thesis.

C. MICROPROCESSOR PASCAL TOOL SET

The software development system selected for use is called Micro Processor Pascal (MPP), and was developed by Texas Instruments for use with the 1750A/F9450 microprocessors. It is a complete tool set for software development, allowing software for the 1750A/F9450 to be developed on a VAX-11/780 minicomputer targeted for final use on the 1750A/F9450 microprocessor. The tool set utilizes a superset of standard ANSI Pascal, and adds facilities to use the 1750A's multiprocessor/multitasking capabilities. The tool set includes a compiler, an assembler, a binder and linker, a reverse assembler (to generate assembly code from the compiler output) and a debugger-simulator. The components and operation of the tool set is shown in Figure 4 [Ref. 3]. The Reverse Assembler which is crucial to the work done here, is particularly useful for allowing hand optimization of a program. This manual tuning of code would allow increased speed of operation, for time critical programs, as a skilled programmer normally writes more efficient code than a compiler.

Another optimization feature of the tool set is the ability of the compiler to partially optimize the object code itself during the compilation. This is dependent upon the programmer using certain programming conventions as described in the MPP/1750A User's Manual. For example, it is found to be faster and more efficient to pass parameters to a procedure by reference than by value. Also, the IF-THEN-ELSE statement is faster than a corresponding CASE statement, if the possible paths can be handled by an IF statement. Even the ordering of variables and data types in the declaration portion of the program is found to affect the execution speed. Further details can be found in the User's Manual.

This tool set was used to write, debug, and test Pascal versions of the two image segmentation algorithms studied here. In addition to determining execution speed estimates using the microprocessor tool set, the same algorithms were also compiled and run under VAX Pascal. This was to allow comparison of the relative speed of execution, and compiler code size generated in the two different environments.

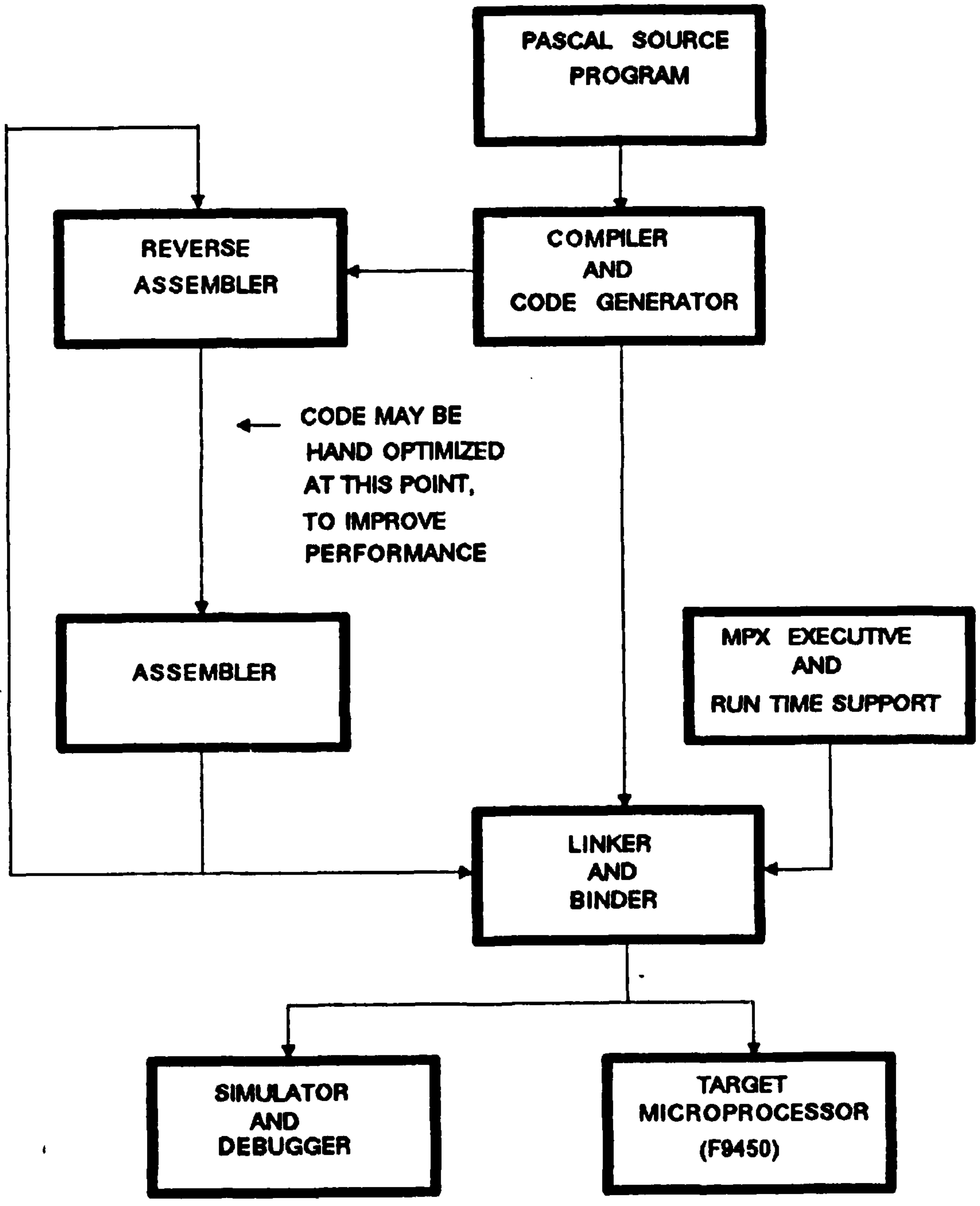

PASCAL SOURCE PROGRAM
REVERSE ASSEMBLER
COMPILER AND CODE GENERATOR
CODE MAY BE HAND OPTIMIZED AT THIS POINT, TO IMPROVE PERFORMANCE
ASSEMBLER
MPX EXECUTIVE AND RUN TIME SUPPORT
LINKER AND BINDER
SIMULATOR AND DEBUGGER
TARGET MICROPROCESSOR (F9450)

Figure 4. Software Development Tool Set

III. METHOD OF EXECUTION SPEED ESTIMATION

A. GENERAL

One of the main purposes of this work was to determine the speed with which the 1750A/F9450 microprocessor could process the image arrays in representative segmentation algorithms (to be described later). Due to a lack of an actual microprocessor system to run the programs on, a method of estimating processor executing speed indirectly had to be found.

As discussed earlier, the tool set Reverse Assembler allows the generation of assembly language programs from the compiled Pascal source code. From this reverse assembled code, it was possible to calculate the total number of executions of a particular instruction (a "JMP" or "CALL" for instance). The Preliminary Data Sheet of the 1750A/F9450 processor contains timing data specifying the amount of time that a given instruction takes to execute. Combining these pieces of information, it is possible to estimate execution times of the assembly language program. The assembled code size (in number of lines of assembly code) was readily obtained by studying the code listings produced by the VAX and Microprocessor Pascal compilers respectively. The speed estimate is of course, not as accurate as actual operational tests on a 1750A/F9450 microprocessor, but it should be a reasonable representation of the processor's performance.

The final results are summarized in Figure 5.

	Program 1	Program 2
Pascal Source Code	462 lines	382 lines
VAX Assembly Code	548 lines	597 lines
MPP Assembly Code	911 lines	1367 lines
VAX Execution Time	8.31-8.41 sec	14.37-14.9 sec
MPP Execution Time	8.78-8.91 sec	14.24-14.8 sec

Figure 5. Summary of Program Test Results:

The two programs, and the meaning of each item in the table will be discussed in detail in Chapter 5, but it can be seen at a glance that the 1750A/F9450 microprocessor should be at least comparable overall in speed to the powerful VAX-11/780 minicomputer. This speaks eloquently of the power of this microprocessor.

B. METHOD OF SPEED EXECUTION ESTIMATE

As to the actual method of execution speed estimation, a brief but representative example is now presented. After a Pascal program has been compiled by the microprocessor tool set, the Reverse Assembler is used to generate an assembly language version of the same program. A small sample of an assembly language program is shown in Figure 6, and will be discussed in this example.

```
       .
       .
LOOOA  EQU  $
       LIM  R2, 00F5A
       CB   R14, 00003
       BLT  L0027
       LIM  R12, 0403E, R13
       LB   R14, 00002
       MSIM R2, 005FB
       AR   R12, R2
       A    R12, 00003, R13
       LR   R4, R12
       STC  0, 00000, R4                     Loop Iteration Path
       LIM  R12, 03B3E, R13
       LB   R14, 00002
       MSIM R2, 005FB
       AR   R12, R2
       A    R12, 00003, R14
       LR   R4, R12
       STC  0, 00000, R4
       INCM 1, 00003, R14
       BR   LOOOA
       .
       .
```

Figure 6. Assembly Language Program Sample

The code shown is a small portion of the assembly language program from one of the two algorithms used. It is a loop, as shown by the arrow, and it is executed a total of 1530 times. From the known number of executions of the loop, the number of each type of instructions contained in the loop, and the timing information from the Preliminary Data Sheet, it is possible to perform the calculations shown in Figure 7.

<u>Type of Instruction:</u>

Load/Store	Add/Subtract	Compare	Jump	Multiply/Divide
LIM: 3	INCM: 1	CB: 1	BLT: 1	MSIM: 2
LR: 2	A: 2		BR: 1	
LB: 2	AR: 2			
STC: 2				
9	5	1	2	2
x .2 uS	x .2 uS	x .4 uS	x .5 uS	x 1.85 uS
1.8 uS +	1.0 uS +	0.4 uS +	1.0 uS +	3.7 uS

= 7.9 uS/iteration of loop

1530 iterations of loop x 7.9 us/iteration of loop

= 3.094 seconds

(Note: the "EQU" takes no execution time.)

Figure 7. Sample Timing Calculation, Based On Figure 6.

As can be seen, the calculation is a relatively simple application of arithmetic, but if based upon accurate timing data, the method should yield reasonably accurate estimates.

The calculations, as already noted, are not difficult, especially for small sections of code as demonstrated. However, for the actual programs, such as those used in this thesis, where there are hundreds of lines of code, the work becomes laborious, and error prone due to miscalculation and other human errors. If this method were necessary for extended use, it might be possible to automate the process, to allow a computer to produce the timing estimates.

Because the calculations here were done by hand, there is a definite possibility of human error, however the calculations were rechecked, so any error should be relatively small. Since the method is only an estimate of the execution speed, it is expected that there will be some errors inherent in the method.

IV. IMAGE SEGMENTATION ALGORITHMS

A. GENERAL

As the purpose of this work is to study the effectiveness of the 1750A/F9450 microprocessor in implementing Image Segmentation, two representative methods of segmentation were selected for testing. Both methods yield similar outputs for similar input data, but use different algorithms to process the input data. Both methods were written in Microprocessor Pascal, and the Pascal listings of each program are included in Appendix 1. The two methods will hereafter be referred to as Programs 1 and 2.

In order to compare and contrast the actual algorithms most accurately, the two programs share as many procedures as possible. Among other procedures, the two programs share identical input and output procedures.

B. PROGRAM 1 OPERATION

Program 1 uses a relatively simple threshold scheme. The input data array is read from a disk file into the program's data array for processing. This image data array is 256 rows by 64 columns in size, and each element of the array is a byte (an integer between 0 and 255) representing the gray level of a pixel in the input image.

The program is written on the assumption that the image consists of a target positioned near the center of the image, surrounded by background. The program initially measures a histogram of the background intensity values by processing the left and right hand most 16 columns. This histogram is an array of the number of pixels having a given intensity versus that intensity.

Following histogram generation, a value representing the average background intensity distribution, is computed by dividing the sum of all histogram intensity values by the number of intensities having nonzero values in the histogram. Finally, a limit value is generated by multiplying the average background intensity distribution value by an empirical threshold value which is pre-selected by the user.

Once the limit value is computed, it is used to process the input image array into a binary output array of the same dimensions. Each image pixel's intensity is read, and the number of pixels having the same intensity value is determined by checking the histogram. If this number of pixels is greater than or equal to the precalculated limit value, the corresponding binary pixel is set to one. If the number of pixels is less than this limit, the corresponding binary pixel is set to zero. The entire binary image is generated in this fashion, pixel by pixel.

This threshold technique tends to generate a significant number of false target and false background pixels, which will appear as random "noise" in the binary image. To eliminate these false pixels, Program 1 uses a final filtering procedure called "REMOVE". This procedure compares each pixel of the binary array, with those surrounding it. If the center pixel has one value, while the surrounding ones are all of the other value, it is assumed that the center pixel is a false one, and its value is reset to the opposite value.

The entire scheme is dependent on the assumption that the image of the target is brighter overall than the background. However, this assumption could be reversed, by switching the inequality in the conversion process.

C. PROGRAM 2 OPERATION

The second program is similar in overall operation, and data flow, but uses a more sophisticated algorithm to perform the processing. Whereas the first program uses only a single pixel attribute (intensity), to determine whether a pixel is a target or background, the second program (also listed in Appendix 1) uses two attributes: intensity, and a computed quantity called "edge magnitude". The edge magnitude is a value which indicates the likelihood that a pixel is part of an edge, or corner of pixels of similar intensity. This is more probable if the pixel is a part of the target, since the background will tend to be a more unstructured pattern of intensities.

The formula used to compute the edge magnitude of an individual pixel is shown in Figure 8 [Ref. 4]:

```
I1   I2   I3              E0 = |Dx| + |Dy|
                <= 3x3
I8   I0   I4     Pixel    Dx = (I1 + 2I8 + I7) - (I3 + 2I4 + I5)
                 Array
I7   I6   I5              Dy = (I1 + 2I2 + I3) - (I7 - 2I6 + I5)
```

Figure 8. Calculation of Pixel Edge Magnitude.

As shown in Figure 8, each pixel in turn, is viewed as the center of a 3 x 3 array of pixels. The Dx and Dy values are calculated from the surrounding pixel intensities, with the equations shown in Figure 8. The desired edge magnitude E0, is the sum of the absolute magnitudes of Dx and Dy. This computation must be performed for every pixel and will be used in the data processing. This will thus involve a great deal of calculation.

As in the first program, the input image array is divided into a target window and a background remainder, though these windows need not be of the same size and/or shape as those in Program 1. This is shown in Figure 9.

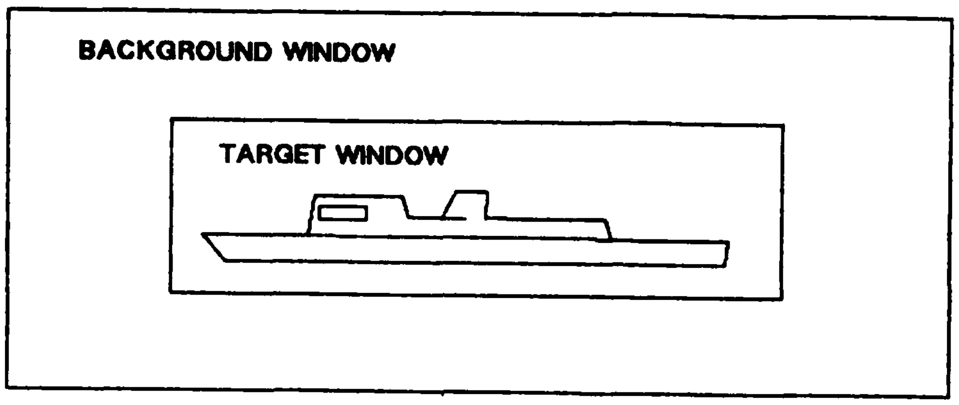


Figure 9. Input Array Target and Background Windows

Program 2 first processes the target window, and each pixel's edge magnitude is calculated. A two dimensional histogram is then developed, containing the number of pixels having each combination of intensity and edge magnitude, versus that combination of intensity and edge magnitude. After completing the target window, the program performs the same operation on the background pixels, generating a separate background histogram.

The program then processes the target window pixels by using a Baysian probability method. For each pixel, the probability of that pixel being a target pixel and of being a background pixel is determined by the use of the target and background histograms. If the target window and background window areas were equal, the probabilities can be read directly off the histograms. If the areas were not equal, the histogram values must be appropriately scaled. The program in this case used equal sized windows, avoiding the need for any scaling.

For each target window pixel, the target and window probabilities are determined from the corresponding histogram values, and are inserted into the following inequality [Ref. 5]:

$$C(B:T)P(X=T) > C(T:B)P(X=B)$$

Where:	$C(B:T)$	is the cost of misclassifying a pixel as a background pixel, if it is a target.
	$P(X=T)$	is the probability that the pixel is a target.
	$C(T:B)$	is the cost of misclassifying a pixel as a target pixel, if it is a background.
	$P(X=B)$	is the probability that the pixel is a background.

If this inequality is true, the pixel being checked is set to one in the binary array. If the equation is false, the pixel is set to zero.

The two cost factors C(B:T) and C(T:B) are constants that the user preselects. The most appropriate value will depend upon the application, and the input data being processed. One likely situation is to set the two equal in value. If this is done, the minimum number of pixels will be misclassified, though there will still be some misclassifications. In Figure 10, the same image was processed, but the cost values were changed for each run, to show the effects of varying these cost values.

In the algorithms used in this work, the analysis was based on the two pixel attributes previously stated. However, the same Baysian probability system can be modified to handle three or more attributes.

One of the advantages of Program 2, is that if the cost factors are properly selected, it generates less of the random noise mentioned earlier, than Program 1. This can eliminate the "Remove" procedure required by Program 1.

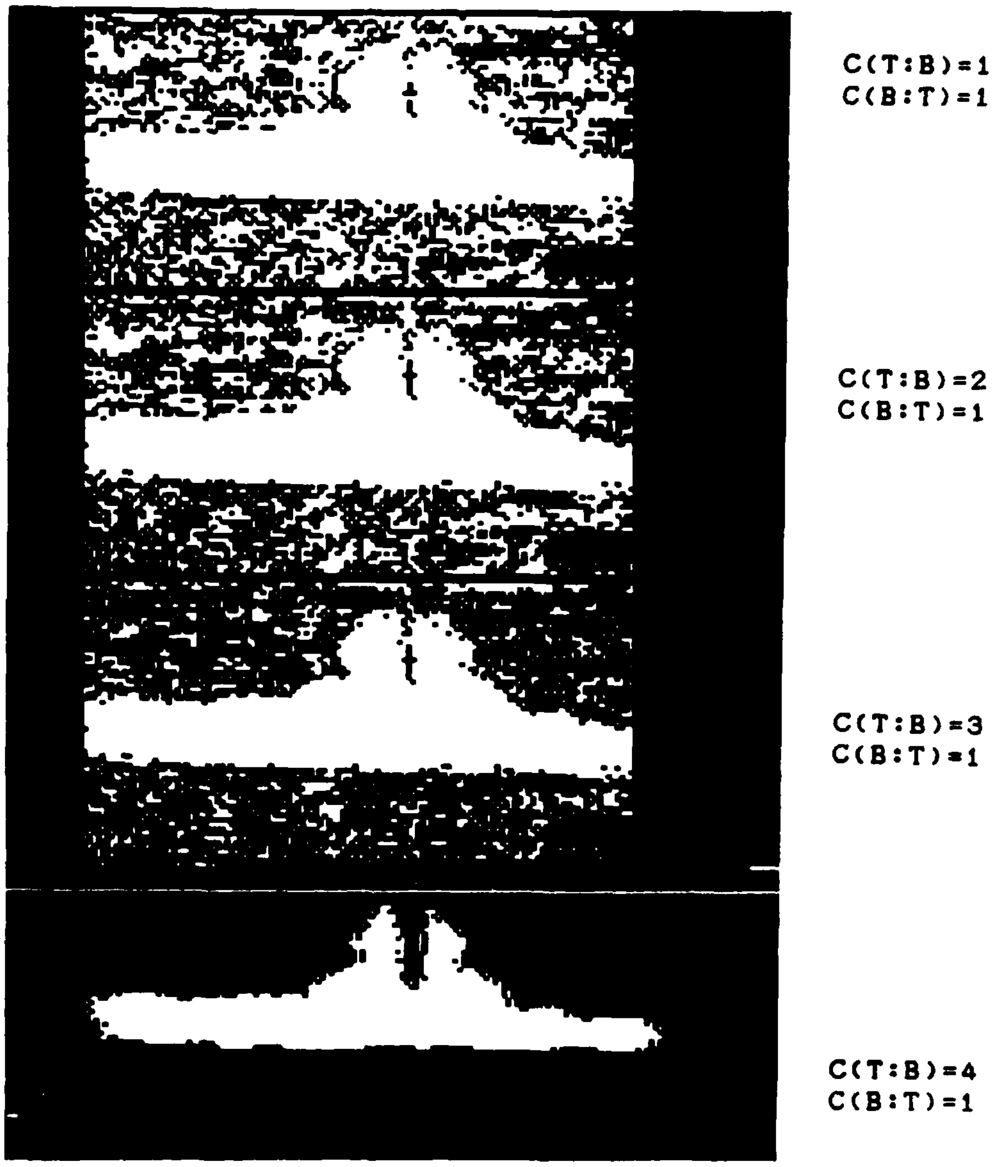


Figure 10. Results of Varying Cost Factors in Baysian Probability (Program 2)

V. ANALYSIS OF THE TEST RESULTS

A. GENERAL

The two image segmentation algorithms discussed were both run for the 1750A/F9450 and VAX systems respectively. The time of execution was actually measured for the VAX system, and calculated for the 1750A/F9450 microprocessor. The code size was determined for each, and all the results were reported in Figure 5. This table is repeated in Figure 11, for easy reference.

	Program 1	Program 2
Pascal Source Code	462 lines	382 lines
VAX Assembly Code	548 lines	597 lines
MPP Assembly Code	911 lines	1367 lines
VAX Execution Time	8.31-8.41 sec	14.37-14.9 sec
MPP Execution Time	8.78-8.91 sec	14.24-14.8 sec

Figure 11. Summary of Program Test Results:

B. COMPARISON OF PROGRAM 1 AND 2

The primary purpose of this study is to determine the applicability of the 1750A/F9450 CPU as to implement image segmentation algorithms. Based upon the MPP and VAX execution times shown in Figure 11, the immediate answer would seem to be that it is indeed, if the VAX itself is adequate. For both Programs 1 and 2, the execution times of the two methods are virtually identical, differing by only a fraction of a second. The fact that the two times are almost identical in this case, suggests that, not only does the Microprocessor Pascal tool set allow the programmer to develop 1750A/F9450 software on a VAX minicomputer, but that program execution times may be estimated by executing the same programs in the VAX Pascal system, rather than calculating them as was done in this work.

It should be noted here, however, that the execution speeds are somewhat variable, as indicated by the range of times in the table. Part of this is due to the variance in input images, which will affect processing time. It would also be affected somewhat, in the VAX case by the presence or absence of a Floating Point Accelerator. The accelerator would not be expected to make a significant difference in this particularly work, because neither program makes extensive use of floating point operations, instead they use byte and integer values.

C. ANALYSIS OF RESULTS

Sheer execution speed is not the sole criterion for determining the value of a given hardware or software system. Other factors can include the support requirements of the hardware, the memory requirements of the software (such as array size, etc.), and any other specialized user needs.

In this study, where the chip used was a version of a military microprocessor (the 1750A), a significant restriction is the memory requirements. This is the case, as the microprocessor might be installed in an aircraft, missile, or other vehicle where space and weight are critical factors. This can limit the amount of physical memory circuitry that can be used, regardless of the amount of logical memory that the microprocessor can actually address.

In image processing, large arrays are normally used to store the image data. One method of attempting to minimize the storage requirements of these arrays is to use "packed arrays" to store data. This can reduce array storage requirements by approximately one half. Packed arrays can have the unfortunately additional effect of increasing execution time, if the system is inefficient in dealing with packed data. The VAX has a variety of data types, that allows efficient implementation of the packed arrays. In particular, there is a Packed Decimal String data type in the VAX system. The 1750A/F9450, unfortunately, does not have such a data type. This requires the Microprocessor Pascal system to use procedures (parts of the run time support library) to pack and unpack the data. This imposed a significant amount of the execution time estimates for the 1750A/F9450, of both Programs 1 and 2. In Program 1, for instance, approximately 3 of 8 seconds of execution time was spent by the 1750A/F9450 system, in packing and unpacking.

Another significant difference in the use of memory, is the size of the program itself. This information is contained in Figure 11, in terms of the number of lines of assembly code for each program, of each system.

In this comparison, the VAX minicomputer has a significant advantage. As shown in Figure 11, the first program had 462 lines of Pascal source code. The VAX system translated this into 548 lines of assembly code, and the 1750A/F9450 required 911 lines of assembly code to do the same thing. This shows that the VAX compiler needed only a 1.19:1 ratio in memory expansion to accommodate the compiled code, while the microprocessor needed a 1.97:1 ratio. For the second program, the ratios were 1.56:1 and 3.58:1.

D. COMPARISON OF VAX AND 1750A/F9450 SYSTEMS

While the two programs produced significantly different ratios between the two systems, the VAX system is consistently on the order of twice as efficient as the

1750A/F9450 microprocessor tool set. This is a significant difference, especially considering the nearly identical execution times. The difference in assembly code size is obviously a matter of concern, since it may be possible to improve the situation, if the cause can be found.

One obvious possibility is the efficiency of the compiler in each system. The VAX system is a commercially available system, and is relatively mature, having gone through the normal revisions as required over a number of years. The 1750A/F9450 tool set is the first version of a microprocessor system, intended largely for military use. Most likely it will be improved in later versions, but this doesn't solve the immediate problem.

This situation may be improved somewhat, by two methods. Firstly, a skilled programmer can take greater care in writing the Pascal version of the program, making it more efficient. It may be possible, for example, to replace a long sequential portion of code, by a shorter loop, which may require less assembly code to implement. Other methods of improvement are those stated earlier, such as improved parameter passing, and the use of IF-THEN-ELSE instead of the CASE statement. Secondly, the Reverse Assembler and Assembler can be used to optimize the assembly code itself. This manually optimized code can then be incorporated into the desired program.

Another significant advantage of the VAX system, is the larger instruction set and types of addressing modes it has, compared to the 1750A/F9450. One very useful addressing mode, shared by the two systems, is known as the Index Addressing Mode by the VAX system, and the Base Relative Mode in the 1750A/F9450. In each case, this mode allows the use of an index register to specify the index of an array entry, thus specifying which element of the array is being addressed. This is shown pictorially in Figure 12 [Ref. 6].

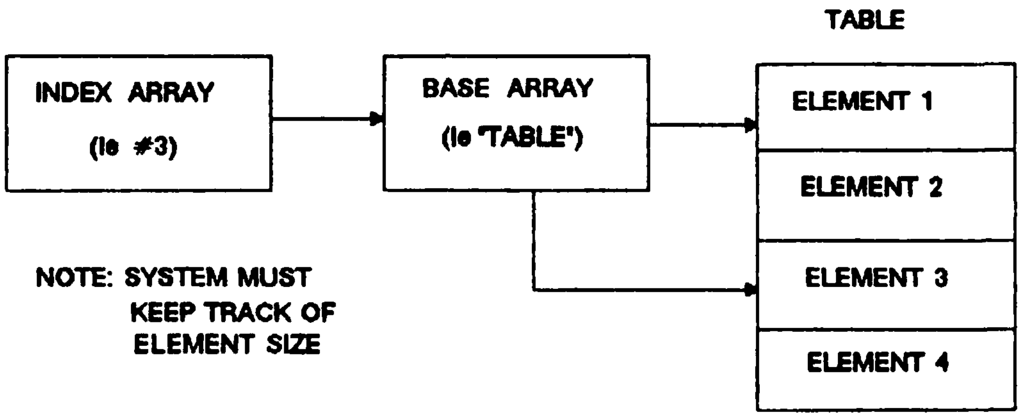


Figure 12. Index Mode/Base Relative Addressing

This mode is particularly useful in array intensive programs, and both of the programs used in this work make frequent use of data arrays. Unfortunately, the 1750A/F9450 Base Relative mode allows only a 256 offset from the base address, which limits its usefulness in this work. The smallest array used in either program contains 16k arrays, which is well beyond the capability of the 256 1750A offset.

The powerful VAX system allows an eight, sixteen or thirty two bit offset values, allowing a potential four gigabyte offset, and can thus easily handle our 16k arrays.

This gives the VAX a significant advantage over the microprocessor. The VAX can handle the array offsets in hardware, while the 1750A/F9450 must do it in software, with the compiler generating a variable to perform this function. This is one instruction which can account for the larger microprocessor assembly code.

As an example of how significant this type of index addressing can be, an example is presented. In Figure 13, a small sample of Pascal is listed, along with the VAX and 1750A assembly code translations of it. The difference in size is obvious, and the reasons for the VAX code being significantly smaller will now be explored.

It is not necessary to have a complete knowledge of assembly code for either system to see that there are significant differences in the manner in which the two systems translate the code. One immediate advantage of the VAX system, is the fact that even at the assembly code level, the system uses the same identifiers as the Pascal source code. This is shown in statements such as "MOVL INFILE,R3". The 1750A/F9450 assembly code on the other hand, uses only register numbers to perform the same function.

Pascal Source Code:

```
FOR I:= 1 TO 64 DO
BEGIN
  FOR J:= 1 TO 256 DO
    IMAGE[I,J]:= INFILE^[J];
END;
```

VAX-11/780

```
1$:  MOVL #1,R12
     NOP
     NOP
2$:  MOVL R12,I
     MOVL #1,R0
     NOP
     NOP
3$:  MOVL R0,J
     INDEX J,#1,#256,#1,#0,R1
     INDEX I,#1,#64,#256,#0,R2
     ADDL2 R1,R2
     MOVL  INFILE,R3
     MOVB -1(R3)[R1],IMAGE-257[R2]
     AOBLEQ #256,R0,3$
     CMPL I,#64
     BGEQ 5$
     PUSHAB INFILE
     CALLS #1,PAS$GET
5$   AOBLEQ #64,R12,2$
     RET
```

1750A/F9450

```
      L R13,LEX$1,R9
      STC 1,0000,R14
L0004 EQU $
      LIM R2,00040
      CB   R14,00000
      BLT L002D
      STC 1,00001,R14
L000A EQU $
      LIM R2,00100
      CB   R14,00001
      BLT L002A
      LIM R12,00005,R13
      LB   R14,00000
      SISP R2,1
      SLL R2,7
      AR   R12,R2
      PSHM R12,R12
      LB   R14,00001
      PSHM R2,R2
      L    R2,04105,R13
      LR   R12,R2
      LB   R12,00000
      PSHM R2,R2
      LB   R14,00000
      SISP R2,1
      SLL  R2,8
      AB   R14,00001
      SISP R2,1
      POPM R3,R3
      CALL LDPI$8
      POPM R4,R4
      SISP R4,1
      POPM R3,R3
      CALL STPI$8
      INCM 1,00001,R14
      BR L000A
L002A EQU $
      INCM 1,00000,R14
      BR L0004
L002D EQU $
      END
```

Figure 13. Comparative Assembly Code Translations

Upon study of Figure 13, it is possible to find some of the reasons for the shorter VAX code.

In the VAX code, three lines allow the use of the powerful VAX Index Addressing Mode. The two lines starting with "INDEX", allow the generation of values in R1 and R2 of the positions of the desired data element based upon an input index (I or J), an offset value (0 here), and the data element size in bytes (1 in this case). More succinctly, for the first INDEX, R1 = (0+J)*1, and for the second, R2 =(0+I)*256. These two values are added, and used as the index to address the infile array. The line to use the index, is MOVB -1(R3)[R1],IMAGE-257[R2]. This line instructs the system to move a byte offset from the first element of INFILE, held in R3, by the number of bytes held in R1, into the position specified in IMAGE-257[R2].

The VAX code of course, uses nested loops, as indicated, to execute this sequence 16k times. To do this, it makes use of AOBLEQ ("Add one and branch if less than or equal") statements. The actual command to "get" the infile, is the CALLS #1,PAS$GET which makes use of a system call.

The code generated by the 1750A/F9450 system is neither short, nor easy to understand, as it makes use of a more primitive set of assembly instructions. As indicated by Figure 13, almost as much code is devoted to maintaining track of the nested loop indices, as the VAX uses for the entire operation. The loop counters are maintained in two locations in memory, R14 + 0000 and R14 + 0001 respectively. These are the locations determined by the contents of R14, offset by zero and one byte. The portion of code within brackets, is the code concerned with actually reading the infile data into the image.

The study of Figure 13 will show that the 1750A must use three separate PUSH's onto the stack (PSHM's) and three separate POP's (POPM's) to produce the addresses necessary to identify the desired infile and image bytes to read and write. This is because the 1750A, as stated earlier, can only offset a maximum of 256 from a specified starting point. To overcome this, the code must "manually" generate the desired indices, by reading the aforementioned R14+0000 and shifting the high order bytes left (the "SLL" commands) and manually adding terms to produce the needed terms.

The "CALL LDPI$8" and "CALL STPI#8" lines are the system calls required to allow the 1750A system read bytes from a packed array ("INFILE") and write bytes to another packed array ("IMAGE").

In general then, it can be seen that the capability to directly operate on larger array indices directly, would significantly improve code size in the 1750A/F9450 system, and could also improve processing time. This would be even more significant for systems using larger arrays than are used here.

One useful addressing mode possessed by the VAX system, but not by the 1750A/F9450, is the Auto Increment/Decrement mode. In this mode, the system automatically increments or decrements the loop index, as required. This is particularly useful in the programs used here, as both algorithms use large numbers of loops, and nested loops in particular. Because of this, any technique such as Auto Increment/Decrement, is bound to improve the speed with which either Program 1 or 2 will execute. Unfortunately, unlike code optimization, new addressing modes cannot be readily implemented into an existing system such as the 1750A/F9450 microprocessor. Thus, this particular shortcoming cannot be easily remedied. The addition of a Memory Management Unit, such as the aforementioned F9451 could impair memory access times, and thus degrade the situation further.

E. SUMMARY

In summary, the 1750A/F9450 would appear comparable overall to the VAX minicomputer in image segmentation speed, but not in the amount of memory needed to implement such algorithms.

Assuming the memory requirements of the 1750A/F9450 microprocessor were not objectionable for a given installation, the next decision would be to determine what the maximum allowable time for processing an image could be.

This would of course be dependent upon the application being used, so it is not possible to give a hard and fast answer as to the applicability of the 1750A/F9450. Some general guidelines may be given however.

If Figure 11 is reviewed it can be seen that, using the tested algorithms, the best processing time would be with the first program, and that approximately 8.78 seconds is required. If the code were highly optimized at both the Pascal and assembly code levels, it is reasonable to expect perhaps a 10% improvement in this. This would result in approximately a 7.9 second conversion time.

If the image being processed were "off-line", such as a medical x-ray, or certain industrial quality control applications, the wait of eight seconds might not be objectionable. This might also be true for some military applications such as a long range sonar, where the signal itself may take something on the order of seconds to reach a target and return.

Many applications however, such as a missile sensor or a pilot's "heads up display" require a much faster processing of data. It would not be reasonable for a pilot to expect his sensors to take eight seconds to update, as a target might very well move out of range in that time.

If it is necessary to attempt to use the 1750A/F9450 in a role such as real time image segmentation, some way must be found to speed up the processing.

VI. CONCLUSIONS AND RECOMMENDATIONS

A. PROBLEMS ENCOUNTERED WITH TOOL SET

The software tool set used in this work is a powerful system. Like any system, it is not perfect however, and some difficulties were encountered.

One problem surfaced when we attempted to compile program 1. The compiler, as would be expected, has a number of default settings which control the compilation unless altered by the user. While these default settings caused no true problems, the user must be aware of these settings [Ref. 7]. First, the system defaults to a 72 column maximum setting. This can cause numerous error messages if a program is transported from a system which uses a standard 80 column line, until the compiler default is changed.

Another default which could cause some problems unless changed, is the fact that the tool set compiler does not routinely check array indices for out of bound conditions unless this feature is specifically activated. This is a helpful feature for such array intensive programs as image segmentation, and the user should be aware that this feature is normally off.

More significantly, the Microprocessor Pascal tool set deals somewhat differently with certain standard Pascal procedures than might be expected [Ref. 8]. It was discovered for example, that to open a disk file, such as the image files used, one needed to use not the expected "OPEN" procedure, but instead either "RESET" or "REWRITE" alone. It was discovered that these procedures both open and reset files for read and write operations. It was also discovered that the procedure "CLOSE" is an external procedure, and must be declared as such.

The next difficulty occurred when program 1 had been successfully compiled. When it was attempted to link the program, numerous error messages were generated, indicating that the system was unable to locate a series of procedures required by the main program. These procedures, bearing such names as F$GET and L$RD, were not user created, and it was found that they were supposed to be part of the system's Run Time Support library. The library was checked, and they were indeed not included.

At first it was feared that the missing procedures had somehow been accidentally erased or destroyed. Upon further study however, it appeared that all of these procedures were involved with the input or output of program data. This appeared to be the case, since the names could be mnemonics for such operations as "file get" and "line read".

After contacting development personnel at Texas Instruments, it was determined that the procedures were intentionally missing. The 1750A/F9450 microprocessor was intended for applications in a wide variety of applications, and thus would need to interface with a wide variety of peripheral equipment, disk drives, terminals, and even real time systems such as sensors. Because of this, it was necessary to keep the 1750A/F9450 as device independent as possible. To do this, the input/output routines were not implemented (though the names such as F$GET were). This would allow (in fact require) the user to develop the routines necessary to perform input/output operations with the user's particular equipment.

It was the lack of input/output capabilities in the tool set as well as the lack of an 1750A/F9450 hardware development system, that dictated the need to develop a means of determining 1750A/F9450 execution speeds indirectly. Even if these routines were in place however, the speed with which a Microprocessor Pascal program ran on a VAX minicomputer would not be expected to be the same as on an 1750A/F9450 microprocessor.

B. METHODS OF IMPROVING EXECUTION SPEED

As described previously, the 1750A/F9450 was found to be too slow in execution speed for real time applications. Therefore, if it is still necessary to use an F9450 or 1750A microprocessor in real time image processing, it will be necessary to find some method of increasing either its speed or the system's actual throughput.

If, in a given application, memory limits are not a problem, a significant improvement could be made in execution time by using "unpacked" arrays instead of "packed" arrays. The data arrays used in the tested programs were 16 kilobytes in packed size. These would approximately double in size if unpacked. If the 1750A/F9450 in a given installation could use multiple megabyte sized memory, it would be feasible to use such unpacked arrays, and thus speed up processing significantly. In program 1 for example, the execution time would go from approximately 8 to approximately 5 seconds, based upon the execution time estimates. (Due to the elimination of packing/unpacking times.)

Another option to speed up processing, is to make use of multiple processors. This could be done in two possible ways: operate the processors in parallel, or operate them in series. Each of these choices offer different methods of improving the processing time.

In studying the operation of the two programs, (as listed in Appendices A and B) it becomes apparent that there are two main operations involved: histogram generation the background and target attributes of the pixels, and generating the binary output arrays based on these histograms. In some cases, it may be possible to perform these operations by two different processors. If the processors are working on the same operation, they are said to be working in parallel. If the processors are processing different operations, they are working in series, which is sometimes also referred to as "pipelining".

In parallel operation, as shown in Figure 14, for program 2, one processor might be generating the target window histogram, while the other program generates the background window. As the two windows are often of the same size, this would take almost exactly the same amount of time, and thus divide the total histogram generation time by a factor of two. Following histogram generation, the two processors might also process the binary image in parallel, by perhaps working on different portions of the image at the same time.

One possible problem with this method, is the difficulty of having multiple processors addressing the same memory simultaneously. If not carefully coordinated, the two processors might attempt to read or write to the same address at the same time. Fortunately, the 1750A/F9450 microprocessor and Microprocessor Pascal tool set are quite well equipped to work in this fashion. In particular, the multitasking capabilities of the Pascal version, and the 1750A/F9450 itself can greatly simplify the coordination of multiple tasks. Additionally, the Memory Management Unit and Block Protect Unit in the 1750A/F9450 chip set can greatly simplify the problem of preventing memory contention.

Another method of preventing memory contention, would be the use of multi port memory. This relatively new technology allows multiple processors to access the same memory simultaneously. Of course, the availability of this technology is not known for all the various applications of the 1750A/F9450.

In the series, or pipelining case, the task of processing the data is also divided between the processors. However, as shown in Figure 15, each processor would perform only one of the functions, either histogram generation or generating the binary image. The first processor would histogram the input image and transfer the histograms to the second processor. The second processor would then use the histograms to generate a binary output image. After each processor is finished, it reads the next input image to perform the same operations.

The pipelining method is somewhat simpler to coordinate than the parallel case, as is not a problem in having two processors attempting to access the same data address simultaneously. It is only necessary to use an interrupt system for each processor to alert the other when it is ready to transfer data from one to the other. This may not speed up the process as much as the parallel case, as the histogram generation may not take the same period of time as the binary generation, so that one processor may sit idle waiting for the other to finish. However, even if the processing of a single image is not as fast as the parallel method, the series method will normally result in a greater total throughput of images. This may be especially useful if the system is continuously processing images, as in the case of a cockpit display for instance.

This pipelining might also be a case were the Built In Function instruction of the 1750A/F9450 microprocessors might be put to use. One processor might "call" the other to generate histograms, and then use them to create output arrays. This would be easier to implement than an elaborate handshaking scheme.

In summary of these two methods, the parallel method will tend to generate a single image more quickly, but the series method will tend to produce a greater total throughput of images. This seems to recommend the parallel method for individual images, and the series scheme for continuously updated image systems.

For maximum improvement, some of these methods could be combined. The same system could make use of improved algorithms, unpacked arrays, and either parallel processing or pipelining. A combination of methods might well reduce the total time for image segmentation to something on the order of one or two seconds. This might well be fast enough for use in some real time systems.

C. SUGGESTIONS FOR FURTHER WORK

Further work remains to be done in several areas. One such area would be to write and implement the necessary input/output procedures needed for the Microprocessor Pascal tool set, for the VAX-11/780 minicomputer system. This would allow much more efficient work with the tool set, than the indirect speed estimates which were done here.

Additional work would also be useful to determine how much improvement might be gained by use of the pipelining and/or parallel schemes described. It might be possible to develop a means of determining exactly when pipelining or parallel processing would be preferable.

Finally, it would be useful to develop an actual 1750A/F9450 hardware system, to allow further work on software development. If such a system becomes available, it would be possible to test the accuracy of the timing calculations done in this thesis.

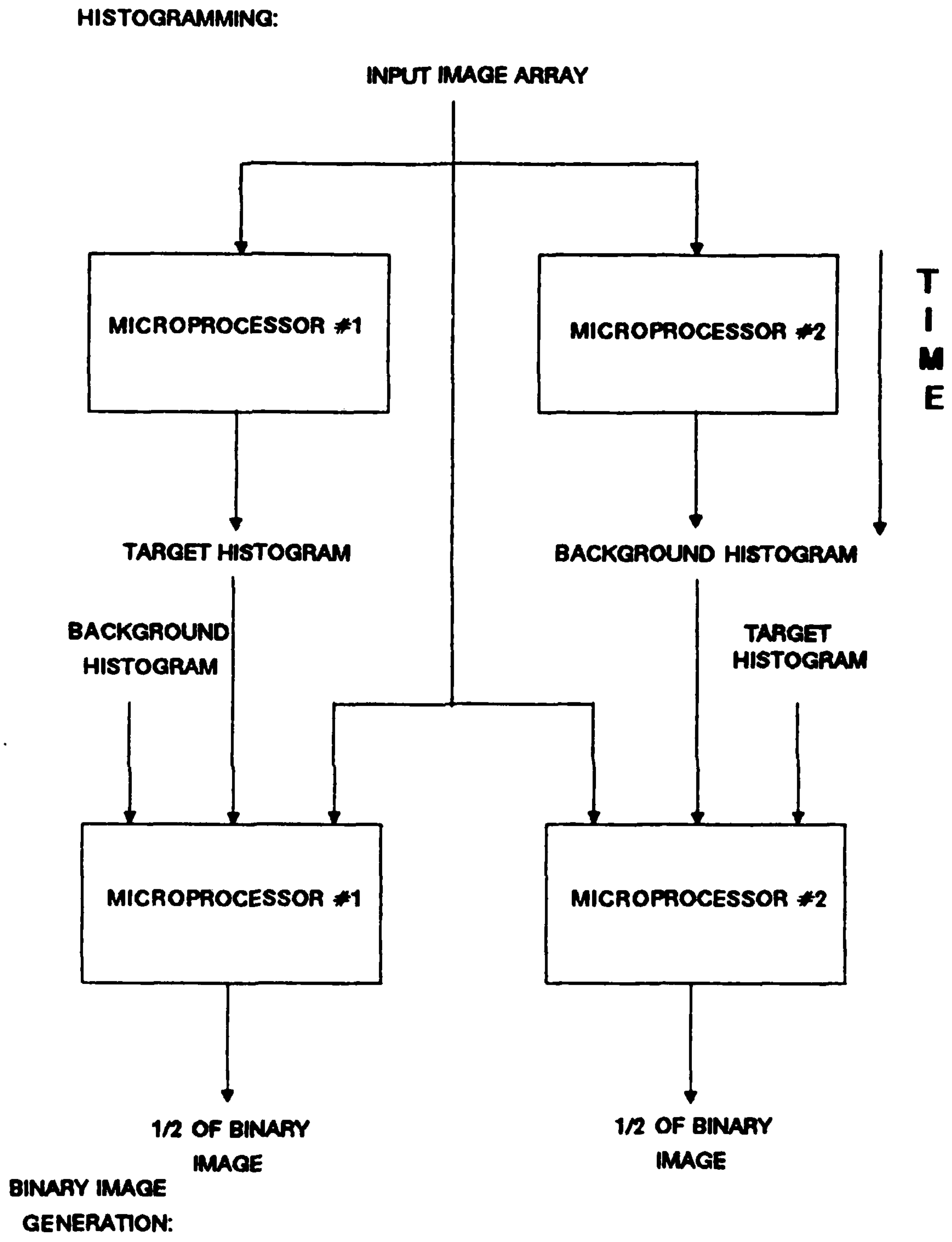


Figure 14. Parallel Processing Scheme for Multiprocessing

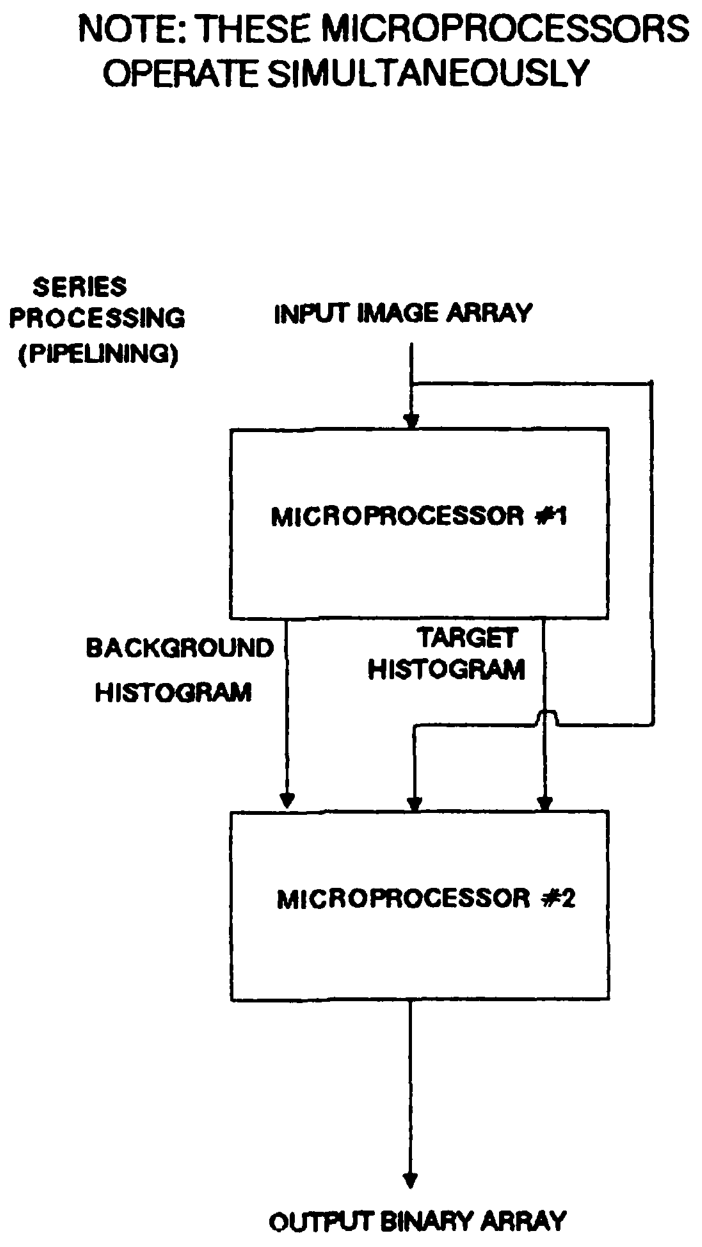


Figure 15. Series Processing Scheme for Multiprocessing (Pipelining)

```
program heuseg(input, output, infile, outfile);

(*****  This program is based on Hughes segmentation program written in
 *****  FORTRAN.   The segmentation algorithm used here was decoded from
 *****  the above Hughes program.
 *****
 *****      Date: Feb, 21, 1985   Author: A. Keiyu
 ****)

const     img_num_col = 256;  (* Number of columns*)
          img_num_row = 64;   (* Number of rows*)


type      byte = 0..255;
          image_array = packed array [1..64,1..256] of byte;
          histogram = array [0..255] of integer;
          long_record = packed array [1..16384] of byte;


var       nth_image : integer;
          gate_col_min, gate_col_max : integer;
          i, j : integer;
          image, binary : image_array;
          perim_hist: histogram;
          infile, outfile : file of long_record;
          infile_name, outfile_name : packed array [1..50] of char;
          getimg_num, imgscl_num, getgate_num: integer;
          segmtb_num, remove_num, wrtseg_num: integer;

(*****    PROCEDURES    *******************)
```

```
(****  GETIMG ************************)
(****  read in an image from the input file *)

 procedure getimg;

   begin


     for i := 1 to img_num_row do
       for j := 1 to img_num_col do
         image[i,j] := infile^[(i-1)*256 + j];

   getimg_num := getimg_num + 1;
   end; (* end GETIMG *)



  (****  IMGSCL **********************)
  (****    add DC bias to the image to compensate for poor
   ****    digitization    *)

   procedure imgscl;

   begin

     (* if a pixel value is less than 255, add 30 to it *)
     for i := 1 to img_num_row do
       for j := 1 to img_num_col do
         if image[i,j] < 225 then image[i,j] := image[i,j] + 30;
   imgscl_num := imgscl_num + 1;
   end; (* end IMGSCL *)
```

```
(****  GETGATE **********)
(****    set a "gate" or a subimage size in the image as specified
 ****    by input parameters.  Also generates a histogram of the
 ****    perimeter of the gate for use in subroutine for segmentation
 ****    computation   *)

 procedure getgate;

 var  perim_col : integer;

 begin


   perim_col := img_num_row div 4;
   gate_col_min := perim_col + 1;
   gate_col_max := img_num_col - perim_col;


  (* clear perimeter histogram *)
   for i := 0 to 255 do
     perim_hist[i] := 0;


  (* histogramming *)

   for i := 1 to img_num_row do
   begin


     (* tally the left edge (16 pixels) *)

     for j := 1 to perim_col do
       perim_hist[image[i, j]] := perim_hist[image[i, j]] + 1;
```

```
    (* tally the right edge (16 pixels) *)

    for j := img_num_col downto img_num_col - perim_col + 1 do
      perim_hist[image[i, j]] := perim_hist[image[i, j]] + 1;


   end; (* end for j *)


  getgate_num := getgate_num + 1;
  end; (* end GETGAT *)



(****  SEGMTB  ********)
(****    compute threshold values based on the perimter histogram
 ****    & segment the image *)

 procedure segmtb;

 var   percenti: real;
       sum_perim_hist, num_bin, limit, ave_distrib : integer;



 begin  (* begin SEGMTB *)


  (* clear sum of perimeter histogram & number of bins *)

   sum_perim_hist := 0;
   num_bin := 0;
```

```
for i := 0 to 255 do
begin

  (* sum the perimter histogram *)

  sum_perim_hist := sum_perim_hist + perim_hist[i];

  (* calculate # of bins containing more than 0 *)

  if perim_hist[i] <> 0 then num_bin := num_bin + 1;

end; (* end for i *)


(* calculate average distribution *)

ave_distrib := sum_perim_hist div num_bin;
percenti := 2.0;

limit := round(percenti/100*ave_distrib);

if limit < 1 then limit := 1;


(* calculate the thresholds *)

for i := 1 to img_num_row do
  for j := gate_col_min to gate_col_max do
```

```
          if perim_hist[image[i, j]] < limit
            then binary[i, j] := 1
            else binary[i, j] := 0;


      (* fill all Os in the both 16-pixel edges *)

      for i := 1 to img_num_row do
      begin

       for j := 1 to gate_col_min - 1 do
         binary[i, j] := 0;

       for j := gate_col_max + 1 to img_num_col do
         binary[i, j] := 0;

      end;

    segmtb_num := segmtb_num + 1;
    end; (* end SEGMTB *)




 (***  REMOVE  ****************)
 (***    Post smoothing procedue used after thresholding  *)
 (***    selectively smooth and image given two thresholds.
  ***    Isolated pixels with a different clustering from surrounding
  ***    pixels will be replaced by surrounding pixel average.  This
  ***    process removes the "salt and pepper" effect which a thresholded
  ***    image with noise can produce. *)

  procedure remove;
```

```
var   smth_wind, half_smth_wind, level, perim_sum, k, 1 : integer;

begin

   smth_wind := 5;
   half_smth_wind := smth_wind div 2;

   for i := (1 + half_smth_wind) to (img_num_row - half_smth_wind)
   do
     for j := (gate_col_min + half_smth_wind) to
                                (gate_col_max - half_smth_wind) do

   begin


     level := 1;


     repeat


       (* initialize perim_sum to 0 *)

       perim_sum := 0;


       (* sum pixel values on the upper/right perimeters *)

       for k := -level to level-1 do
       begin

         perim_sum := perim_sum + binary[i-level, j+k];

         perim_sum := perim_sum + binary[i+k, j+level];

       end;
```

```
 (* sum pixel values on the bottom/left perimeters *)

 for k := -level + 1 to level do
 begin

   perim_sum := perim_sum + binary[i+k, j-level];

   perim_sum := perim_sum + binary[i+level, j+k];

 end;


(*  perimeter pixels are all 0s *)
(*  then if a pixel in the center area is 1, set it to 0
    & escape from the repeat loop by setting level to more then
    half_smth_wind *)

 if perim_sum = 0 then
   for k := -(level-1) to (level-1) do
     for l := -(level-1) to (level-1) do
       if binary[i+k, j+l] = 1 then
       begin
         binary[i+k, j+l] := 0;
         level := smth_wind;
       end;


(* perimeter pixels are all 1s *)
(* then if a pixel in the center area is 0, set it to 1
   and escape from the repeat loop in the same manner
*)
```

```
    if perim_sum = level*8 then
     for k := -(level-1) to (level-1) do
      for l := -(level-1) to (level-1) do
      if binary[i+k, j+l] = 0 then
      begin
       binary[i+k, j+l] := 1;
       level := smth_wind;
      end;


     (* increment level with 1 *)

   level := level + 1


  until (level > half_smth_wind);


 end (* end for i & j *)

remove_num := remove_num + 1;
end; (* end REMOVE *)





(****  WRTSEG *******)
(****    write the segmented image (binary) to the disk
*)
```

```
  procedure wrtseg;

  begin

    for i := 1 to img_num_row do
      for j := 1 to img_num_col do
        outfile^[(i-1)*256 + j] := binary[i, j];

    put(outfile);

   wrtseg_num := wrtseg_num + 1;
   end; (* end WRTSEG *)



(***********              MAIN              PROGRAM              ****************)
(********************************************)

    begin (* initialize all procedure counters*) getimg_num := 0; imgscl_num  :=
0; getgate_num := 0; segmtb_num := 0; remove_num := 0; wrtseg_num := 0;

     (***   read in input file name/output file name *)
      write(' Input image file name => ');
      readln (infile_name);


      write(' Output image file name => ');
      readln (outfile_name);


     (***  open input & output files ***)
      open (infile, infile_name, history := old, access_method := direct
      ,
      record_length := 16384, record_type := fixed);
 reset (infile);


 open (outfile, outfile_name, history := new, access_method := dire
      ct,
      record_length := 16384, record_type := fixed);
 rewrite (outfile);



(*** initialize "nth_image" to O  ***)
 nth_image := O;



(***  read in an image, process it & write its result back until
     end-of-file is detected   ***)

repeat


  (***  load an image from infile array(16384)
        into image array (64x256) ***)
   getimg;



  nth_image := nth_image + 1;(* count # of images *)
  writeln('processing image', nth_image);




  (***  the last line of the image is used to carry ship information
      therefore, replace the last line with the second last line
      so that the last line doesn't interfere with segmenting
*)
  for j := 1 to 256 do
    image[64, j] := image[63, j];


(***  add DC bias  ***)
 imgscl;


(***  get a 'gate' ***)
 getgate;


(***  threshold & segmentation  ***)
 segmtb;


(*** post smoothing  ***)
 remove;


(***  write the result to the disk ***)
 wrtseg;


(*** get the next image from the infile *)
 get(infile);
 until eof(infile);


 (*** close input & outputfile ***)
  close(infile);
  close(outfile);
 (* output procedure counters *)
writeln('getimg = ',getimg_num);
writeln('imgscl = ',imgscl_num);
writeln('getgate = ',getgate_num);
writeln('segmtb = ',segmtb_num);
writeln('remove = ',remove_num);
writeln('wrtseg = ',wrtseg_num);


end.  (* end of the main program *)
```

```
program bay3(input,output,infile,outfile);

(*****  This program is based a on Hughes segmentation program written in
 *****  FORTRAN, but has been modified to make use of Baysian probability.
 *****
 *****
 *****      Date: Nov 10, 1985     Author: Percy D. Cody III
 ****)

const     img_num_col = 256; (* Number of columns*)
          img_num_row = 64;  (* Number of rows*)


type      byte = 0..255;
          image_array = packed array [1..64,1..256] of byte;
          histogram = array [0..255,0..1530] of integer;
          long_record = packed array [1..256] of byte;


var       nth_image : integer;
          gate_col_min, gate_col_max : integer;
          i, j : integer;
          image,binary: image_array;
          infile,outfile: file of long_record;
          infile_name, outfile_name : packed array [1..50] of char;
          getimg_num,imgscl_num,pfeat_num: integer;
          bclass_num,wrtseg_num:                integer;
target_hist,back_hist:histogram;

(*****    PROCEDURES   *******************)

  PROCEDURE CLOSE(VAR F: ANYFILE); EXTERNAL;
```

```
(****  GETIMG ************************)
(****  read in an image from the input file *)

 procedure getimg;

   begin


     for i := 1 to img_num_row do
       begin
       for j := 1 to img_num_col do
         image[i, j] := infile^[j];
       if i< 64 then get(infile);
     end;
   getimg_num := getimg_num + 1;
   writeln('getimg');
   end; (* end GETIMG *)




  (****  IMGSCL ***********************)
  (****    add DC bias to the image to compensate for poor
   ****    digitization    *)

   procedure imgscl;

   begin

     (* if a pixel value is less than 225, add 30 to it *)
     for i := 1 to img_num_row do
       for j := 1 to img_num_col do
         if image[i, j] < 225 then image[i, j] := image[i, j] + 30;
   imgscl_num := imgscl_num + 1;
   writeln('imgscl');
   end; (* end IMGSCL *)
```

```
    (*** PFEAT ***)
    (*** compute the edge magnitude of each target and feature
     *** pixel, and compute the background and target histo-
     *** grams of the desire features ***)

    procedure pfeat;           var  dx,dy,i,j:  integer;                   intensity,
edge_mag: integer;

    begin           for  i  :=  0  to  255  do  (*  zero  histogram  arrays  *)
               for  j  :=  0  to 1530 do                      begin
target_hist[i,j]    :=    0;                              back_hist[i,j]    :=    0;
               end; (* zeroing *)

    for i := 2 to 63 do (* compute left background edge_mag *)
      for  j  :=  2  to  47  do                  begin                        dx   :=
image[i+1,j-1]   +   2*image[i,j-1]   +   image[i-1,j-1]                              -
image[i+1,j+1]  -  2*image[i,j+1]  -   image[i-1,j+1];                        dy   :=
image[i+1,j-1]  +  2*image[i+1,j] + image[i+1,j+1]                       - image[i-
1,j-1] - 2*image[i-1,j] - image[i-1,j+1];                 edge_mag := abs(dx)  +
abs(dy);                back_hist[image[i,j],edge_mag]   :=
back_hist[image[i,j],edge_mag] + 1;                                  end;

    for i := 2 to 63 do (* compute right background edge_mag *)         for j :=
178   to  255  do               begin                dx  :=  image[i+1,j-1]  +
2*image[i,j-1]  +   image[i-1,j-1]                     -   image[i+1,j+1]   -
2*image[i,j+1]   -   image[i-1,j+1];                  dy  :=  image[i+1,j-1]  +
2*image[i+1,j]  +   image[i+1,j+1]                     -   image[i-1,j-1]   -
2*image[i-1,j]  - image[i-1,j+1];             edge_mag := abs(dx) + abs(dy);
               back_hist[image[i,j],edge_mag]          :=
back_hist[image[i,j],edge_mag] + 1;            end;

    for i := 2 to 63 do (* compute target edge_mag *)         for j := 49 to 172
do
          begin                 dx  :=  image[i+1,j-1]  +  2*image[i,j-1]   +
image[i-1,j-1]                   - image[i+1,j+1] - 2*image[i,j+1] - image[i-
1,j+1];
```

```
                dy   :=  image[i+1, j-1]   +   2*image[i+1, j]   +   image[i+1, j+1]
                     -   image[i-1, j-1]   -   2*image[i-1, j]   -  image[i-1, j+1];
                edge_mag          :=          abs(dx)          +           abs(dy);
                target_hist[image[i, j], edge_mag]           :=
target_hist[image[i, j], edge_mag] + 1;                    end;

    pfeat_num := pfeat_num + 1;
    writeln('pfeat');
    end;  (* pfeat *)

    (*** BCLASS ***)
    (*** procedure to use baysian probability to classify
         pixels as target or background ****)

    procedure bclass;          const cost1 = 3.0; (*  cost  of  misclassifying  a
target  pixel  *)                   cost2 = 1.0; (* cost of misclassifying a back-
ground pixel *)                alpha = 0.75; (* % of background in target  window
*)

        var     a:    real;                     ht, hb, dy, dx:    integer;
intensity, edge_mag:  integer;

        begin          a := (cost1+cost2)/cost2;            for i:= 1 to 64 do {
zero  binary image }           for j:= 1 to 256 do                  binary[i, j]
:= 0;                for i := 2 to 63 do (* process target window *)
for  j  :=  49  to  172  do                   begin               intensity  :=
image[i, j];                  dx := image[i+1, j-1]  +  2*image[i, j-1]  +  image[i-
1, j-1]                        - image[i+1, j+1] - 2*image[i, j+1] - image[i-1, j+1];
                dy   :=  image[i+1, j-1]   +   2*image[i+1, j]   +   image[i+1, j+1]
                  - image[i-1, j-1] - 2*image[i-1, j] - image[i-1, j+1];
                edge_mag   :=   abs(dx)   +   abs(dy);                    ht   :=
target_hist[intensity, edge_mag];                              hb             :=
back_hist[intensity, edge_mag];                 if (ht  =  0)
then   binary[i, j]:=    0                      else                      begin
                  if (ht > (alpha*a*hb))                     then  binary[i, j]
:=  255                   else binary[i, j] := 0;                   end; {else}
```

```
end; (* target window *)


          for i:= 65 to 192 do (* zero top and bottom lines *)             begin
                binary[1,i]  :=  0;                           binary[64,i] := 0;
end; (* zero lines *)

        bclass_num := bclass_num + 1;
        writeln('bclass');            end; (* bclass *)


    (****  WRTSEG *******)
    (****    write the segmented image (binary) to the disk
    *)
     procedure wrtseg;

     begin
       for i := 1 to img_num_row do
         begin
         for j := 1 to img_num_col do
           outfile^[j] := binary[i,j];
         put(outfile);
         end;
      wrtseg_num := wrtseg_num + 1;
      end; (* end WRTSEG *)


(***********             MAIN              PROGRAM               ****************)
(*******************************************)

    begin (* initialize all procedure counters*) getimg_num := 0; imgscl_num  :=
0; pfeat_num  := 0; bclass_num := 0; wrtseg_num := 0;


     (***   read in input file name/output file name *)
```

```
 write(' Input image file name => ');
 readln (infile_name);


 write(' Output image file name => ');
 readln (outfile_name);


(***  open input & output files ***)
 reset (infile);


 rewrite (outfile);


(*** initialize "nth_image" to 0  ***)
 nth_image := 0;



(***  read in an image, process it & write its result back until
     end-of-file is detected   ***)

repeat


  (***  load an image from infile array(16384)
        into image array (64x256) ***)
   getimg;



 nth_image := nth_image + 1;(* count # of images *)
 writeln('processing image', nth_image);
```

```
(***  the last line of the image is used to carry ship information
      therefore, replace the last line with the second last line
      so that the last line doesn't interfere with segmenting
*)
  for j := 1 to 256 do
    image[64, j] := image[63, j];


(***  add DC bias  ***)
 imgscl;


(*** compute edge magnitudes and create histograms ***)          pfeat;


(*** classify pixels by Baysian probability ***)          bclass;


(***  write the result to the disk ***)
 wrtseg;


(*** get the next image from the infile *)
 get(infile);


until eof(infile);


(*** close input & outputfile ***)
```

```
  close(infile);
  close(outfile);


 (* output procedure counters *)
writeln('getimg = ',getimg_num);
writeln('imgscl = ',imgscl_num);
writeln('pfeat_num = ',pfeat_num);
writeln('bclass_num = ',bclass_num);
writeln('wrtseg = ',wrtseg_num);


end.  (* end of the main program *)
```

LIST OF REFERENCES

1. Fairchild Camera and Instrument Corporation, Microprocessor Division, F9450 High-Performance 16-Bit Bipolar Microprocessor, Preliminary Data Sheet - July 1984, pp. 2, July 1984.

2. Fairchild Camera and Instrument Corporation, Microprocessor Division, F9450 High-Performance 16-Bit Bipolar Microprocessor, Preliminary Data Sheet - July 1984, pp. 16, July 1984.

3. Texas Instruments Corporation, MPP/1750A USER'S MANUAL, pp. 1-2, Release 1.3.

4. Duda, R. O., and Hart, P. E., Pattern Classification and Scene Analysis, pp. 267-272, John Wiley and Sons, Inc., 1973.

5. Duda, R. O., and Hart, P. E., Pattern Classification and Scene Analysis, pp. 13-17, John Wiley and Sons, Inc., 1973.

6. Levy, H. M. and Eckhouse, R. H. Jr., Computer Programing and Architecture, the VAX-11, 1st ed., pp. 80, Digital Press, April 1980.

7. Texas Instruments Corporation, MPP/1750A USER'S MANUAL, pp. 2-9 - 2-10, Release 1.3.

8. Texas Instruments Corporation, MPP/1750A USER'S MANUAL, pp. D-1 - D-2, Release 1.3.

INITIAL DISTRIBUTION LIST

		No. of Copies
1.	Library, Code 0142 Naval Postgraduate School Monterey, California 93943-5100	2
2.	Defense Technical Information Center Cameron Station Alexandria, Virginia 22304-6145	2
3.	Department Chairman, Code 62 Department of Electrical and Computer Engineering Monterey, California 93943	1
4.	Professor Chin-Hwa Lee, Code 62Le Department of Electrical and Computer Engineering Naval Postgraduate School Monterey, California 93943	2
5.	Professor Alex Gerba Jr., Code 62Gz Department of Electrical and Computer Engineering Naval Postgraduate School Monterey, California 93943	1
6.	Commanding Officer Naval Ocean System Center Attn: Lt. Percy D. Cody III San Diego, California 92152	1
7.	Commanding Officer Naval Ocean System Center Attn: C. E. Holland Jr. Code 811 San Diego, California 92152	1
8.	Commanding Officer Naval Ocean System Center Attn: Mike Stelmach Code 811 San Diego, California 92152	1
9.	Commanding Officer Naval Ocean System Center Attn: James Wasson Code 811 San Diego, California 92152	1